

Tcp7-7

采用主从模型（Master-Worker）进行 TCP 服务器开发，使用 epoll 的边缘触发（ET）模式，并使用队列来分配新的连接是一种高效的设计。以下是详细的步骤和示例代码，帮助你实现这个设计。

1. 主从模型的工作原理

- **主线程**：负责监听新的连接请求，并将新的连接分配给工作线程。
- **工作线程**：负责处理连接上的 I/O 操作。

2. 单生产者多消费者队列

在这种模型中，主线程是单生产者，多个工作线程是多消费者。主线程将新的连接插入到队列中，工作线程从队列中获取连接并处理。

3. 维护队列

可以使用一个线程安全的队列来维护新连接的分配。C++ 标准库中的 `std::queue` 和 `std::mutex` 可以用来实现线程安全的队列。

示例代码

主要组件

1. **主线程**：监听新的连接并将其加入队列。
2. **工作线程**：从队列中获取连接并处理。
3. **线程池**：管理工作线程。

代码实现

以下是一个示例实现：

```
cpp
复制代码
#include <iostream>
#include <thread>
#include <vector>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <sys/epoll.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>

#define MAX_EVENTS 10
```

```

#define PORT 8080

void setNonBlocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        throw std::runtime_error("Failed to get file flags");
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        throw std::runtime_error("Failed to set non-blocking flag");
    }
}

class ThreadPool {
public:
    ThreadPool(size_t numThreads);
    ~ThreadPool();
    void enqueue(int fd);
    void run();

private:
    std::vector<std::thread> workers_;
    std::queue<int> tasks_;
    std::mutex queueMutex_;
    std::condition_variable condition_;
    bool stop_;

    void worker();
};

ThreadPool::ThreadPool(size_t numThreads) : stop_(false) {
    for (size_t i = 0; i < numThreads; ++i) {
        workers_.emplace_back(&ThreadPool::worker, this);
    }
}

ThreadPool::~~ThreadPool() {
    {
        std::unique_lock<std::mutex> lock(queueMutex_);
        stop_ = true;
    }
    condition_.notify_all();
    for (std::thread &worker : workers_) {
        worker.join();
    }
}

void ThreadPool::enqueue(int fd) {
    {
        std::unique_lock<std::mutex> lock(queueMutex_);
        tasks_.push(fd);
    }
    condition_.notify_one();
}

```

```

}

void ThreadPool::run() {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        throw std::runtime_error("Failed to create epoll file descriptor");
    }

    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd == -1) {
        throw std::runtime_error("Failed to create socket");
    }
    setNonBlocking(listen_fd);

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(PORT);

    if (bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        throw std::runtime_error("Bind failed");
    }

    if (listen(listen_fd, 10) == -1) {
        throw std::runtime_error("Listen failed");
    }

    struct epoll_event event;
    event.data.fd = listen_fd;
    event.events = EPOLLIN | EPOLLET; // 边缘触发
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &event) == -1) {
        throw std::runtime_error("Failed to add listen file descriptor to epoll");
    }

    std::vector<struct epoll_event> events(MAX_EVENTS);

    while (!stop_) {
        int n = epoll_wait(epoll_fd, events.data(), MAX_EVENTS, -1);
        if (n == -1) {
            throw std::runtime_error("epoll_wait failed");
        }

        for (int i = 0; i < n; ++i) {
            if (events[i].events & EPOLLIN) {
                if (events[i].data.fd == listen_fd) {
                    while (true) {
                        int conn_fd = accept(listen_fd, nullptr, nullptr);
                        if (conn_fd == -1) {
                            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                                break;
                            }
                        } else {
                            throw std::runtime_error("Accept failed");
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
    }
    setNonBlocking(conn_fd);
    enqueue(conn_fd);
}

}

}

}

}

close(listen_fd);
close(epoll_fd);
}

void ThreadPool::worker() {
    while (true) {
        int fd;
        {
            std::unique_lock<std::mutex> lock(queueMutex_);
            condition_.wait(lock, [this] { return stop_ || !tasks_.empty(); });
            if (stop_ && tasks_.empty()) {
                return;
            }
            fd = tasks_.front();
            tasks_.pop();
        }

        char buffer[1024];
        while (true) {
            ssize_t bytes = read(fd, buffer, sizeof(buffer));
            if (bytes > 0) {
                write(fd, buffer, bytes); // Echo back the data
            } else if (bytes == -1) {
                if (errno == EAGAIN) {
                    break;
                } else {
                    close(fd);
                    break;
                }
            } else {
                close(fd);
                break;
            }
        }
    }
}

int main() {
    try {
        ThreadPool pool(std::thread::hardware_concurrency());
        pool.run();
    } catch (const std::exception &ex) {

```

```
        std::cerr << "Error: " << ex.what() << std::endl;
        return 1;
    }
    return 0;
}
```

代码说明

1. `setNonBlocking` 函数：设置文件描述符为非阻塞模式。
2. `ThreadPool` 类
 - ：管理工作线程和任务队列。
 - `enqueue` 方法：将新的连接文件描述符添加到任务队列中。
 - `run` 方法：主线程的主循环，监听新的连接并将其分配给工作线程。
 - `worker` 方法：工作线程，从任务队列中获取连接并处理。
3. `main` 函数：创建线程池并启动主线程的主循环。

维护队列

- 定义：任务队列 `tasks_` 定义在 `ThreadPool` 类中，并由 `queueMutex_` 保护。
- 添加任务：主线程在 `enqueue` 方法中向任务队列中添加新的连接。
- 获取任务：工作线程在 `worker` 方法中从任务队列中获取连接进行处理。

延迟的测量和基准测试

1. 延迟测量：
 - 使用 `tcpdump` 捕获数据包，分析请求和响应的时间戳来测量延迟。
2. 基准测试方案：
 - 使用工具如 `wrk`、`ab` 或 `httpperf` 进行 HTTP 性能测试。
 - 通过逐步增加并发用户数进行压力测试，记录最大吞吐量和延迟。

示例

使用 `wrk` 进行基准测试：

```
sh
复制代码
wrk -t12 -c400 -d30s http://localhost:8080
```

此命令将启动 12 个线程，模拟 400 个并发连接，测试持续 30 秒。

通过这种方式，你可以高效地管理新的连接，并使用线程池提高并发性能。同时，通过详细的延迟测量和基准测试，可以优化系统性能，满足高并发场景的需求。

在应用层数据通过 TCP 传输时，由于 TCP 是流式协议，没有消息边界的概念，这就可能导致粘包和拆包的问题。我们需要在应用层自行区分报文的边界。通常在以下几个地方需要处理和区分 TCP 报文的边界：

序列化和反序列化

1. 序列化

:

- 发送端将应用层数据转换为字节流并添加必要的消息边界标记（如消息长度或分隔符），然后发送给 TCP 缓冲区。

2. 反序列化

:

- 接收端从 TCP 缓冲区读取字节流，根据消息边界标记将字节流转换回应用层数据结构。

处理流程中的报文边界区分

发送数据流程

1. 应用层发数据

:

- 序列化数据，将应用层数据结构转换为字节流。

2. 用户自定义缓冲区

:

- 在发送前将数据放入用户自定义的发送缓冲区，添加消息边界标记。

3. TCP 缓冲区

:

- 调用 `send` 函数将数据从用户缓冲区发送到 TCP 缓冲区。

接收数据流程

1. TCP 缓冲区

:

- 调用 `recv` 函数从 TCP 缓冲区接收数据。

2. 用户自定义缓冲区

:

- 将接收到的数据放入用户自定义的接收缓冲区。

3. 应用层

:

- 从接收缓冲区读取数据，进行反序列化，将字节流转换回应用层数据结构。

示例代码：区分报文边界

发送端代码（添加消息头部）

```
cpp
复制代码
#include <iostream>
```

```

#include <string>
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>

// 序列化并发送消息
void sendMessage(int sockfd, const std::string& message) {
    uint32_t length = htonl(message.size());
    send(sockfd, &length, sizeof(length), 0); // 发送长度
    send(sockfd, message.c_str(), message.size(), 0); // 发送数据
}

int main() {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("socket");
        return 1;
    }

    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    inet_pton(AF_INET, "127.0.0.1", &serverAddr.sin_addr);

    if (connect(sockfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("connect");
        close(sockfd);
        return 1;
    }

    std::string message = "Hello, TCP!";
    sendMessage(sockfd, message);

    close(sockfd);
    return 0;
}

```

接收端代码（解析消息头部）

```

cpp
复制代码
#include <iostream>
#include <vector>
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>

// 反序列化并接收消息
std::string recvMessage(int sockfd) {
    uint32_t length;
    if (recv(sockfd, &length, sizeof(length), MSG_WAITALL) != sizeof(length)) {

```

```

        return "";
    }
    length = ntohl(length);

    std::vector<char> buffer(length);
    if (recv(sockfd, buffer.data(), length, MSG_WAITALL) != (ssize_t)length) {
        return "";
    }

    return std::string(buffer.begin(), buffer.end());
}

int main() {
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket");
        return 1;
    }

    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("bind");
        close(listenfd);
        return 1;
    }

    if (listen(listenfd, 10) == -1) {
        perror("listen");
        close(listenfd);
        return 1;
    }

    int connfd = accept(listenfd, nullptr, nullptr);
    if (connfd == -1) {
        perror("accept");
        close(listenfd);
        return 1;
    }

    std::string message = recvMessage(connfd);
    std::cout << "Received: " << message << std::endl;

    close(connfd);
    close(listenfd);
    return 0;
}

```


序列化和反序列化的位置

- 发送端：序列化在应用层发数据到用户自定义缓冲区之间进行。
- 接收端：反序列化在用户自定义缓冲区到应用层之间进行。

闭包调用的作用

闭包（Closure）是一种函数对象，它捕捉了创建它的环境中的变量，并能够在其内部函数中访问这些变量。闭包的作用在于能够将一个函数及其环境封装起来，使得这个函数可以在其被调用时访问其外部作用域中的变量。

闭包的作用：

1. **保持状态**：闭包可以保持状态信息，这对于实现回调函数、事件处理器等非常有用。
2. **延迟计算**：闭包可以用于延迟计算，例如延迟执行某些操作，直到满足特定条件。
3. **抽象和封装**：闭包提供了一种将函数和其环境封装在一起的方法，提高了代码的抽象层次和封装性。

闭包示例

```
cpp
复制代码
#include <iostream>
#include <functional>

std::function<int(int)> createAdder(int x) {
    return [x](int y) { return x + y; }; // 闭包捕捉外部变量 x
}

int main() {
    auto adder = createAdder(10); // 创建一个加法器
    std::cout << adder(5) << std::endl; // 输出 15
    return 0;
}
```

在这个示例中，`createAdder` 函数返回一个闭包，这个闭包捕捉了外部变量 `x`，并在内部函数中使用它。这样可以实现创建不同的加法器，每个加法器都保持了不同的状态。

设计思路

1. **内存池（MemoryPool）**：管理固定大小的内存块，减少频繁的内存分配和释放。
2. **循环缓冲区（Buffer）**：使用循环缓冲区来存储数据，实现高效的读写操作。
3. **Connection 类**：管理每个用户的连接，包含读缓冲区和写缓冲区，处理数据发送和接收。

具体设计

1. **内存池（MemoryPool）**：
 - 提供固定大小的内存块。

- 用于分配和回收内存块。
- 2. **循环缓冲区 (Buffer)** :
 - 通过 head 和 tail 指针实现循环读写。
 - 避免多次拷贝和内存分配。
- 3. **Connection 类**:
 - 包含读缓冲区和写缓冲区。
 - 通过消息头部携带长度信息来区分不同的消息边界。

详细代码

MemoryPool 类

```
cpp
复制代码
#include <iostream>
#include <vector>
#include <stack>
#include <stdexcept>

// 内存池类，用于管理固定大小的内存块
class MemoryPool {
public:
    MemoryPool(size_t blockSize, size_t poolSize)
        : blockSize_(blockSize), poolSize_(poolSize) {
        // 预分配内存块
        pool_.resize(blockSize * poolSize);
        for (size_t i = 0; i < poolSize; ++i) {
            freeBlocks_.push(&pool_[i * blockSize]);
        }
    }

    // 分配内存块
    void* allocate() {
        if (freeBlocks_.empty()) {
            throw std::runtime_error("Memory pool exhausted");
        }
        void* ptr = freeBlocks_.top();
        freeBlocks_.pop();
        return ptr;
    }

    // 回收内存块
    void deallocate(void* ptr) {
        freeBlocks_.push(static_cast<char*>(ptr));
    }

private:
    size_t blockSize_;
    size_t poolSize_;
    std::vector<char> pool_;
}
```

```
std::stack<void*> freeBlocks_;  
};
```

Buffer 类

```
cpp  
复制代码  
#include <cstring>  
  
// 循环缓冲区类，实现高效的读写操作  
class Buffer {  
public:  
    Buffer(size_t size, MemoryPool& pool)  
        : buffer_(static_cast<char*>(pool.allocate())), size_(size), pool_(pool),  
        head_(0), tail_(0), full_(false) {}  
  
    ~Buffer() {  
        pool_.deallocate(buffer_);  
    }  
  
    // 写入数据到缓冲区  
    size_t write(const char* data, size_t len) {  
        size_t space = capacity() - size();  
        size_t toWrite = std::min(len, space);  
  
        for (size_t i = 0; i < toWrite; ++i) {  
            buffer_[head_] = data[i];  
            head_ = (head_ + 1) % size_;  
        }  
  
        if (head_ == tail_) {  
            full_ = true;  
        } else {  
            full_ = false;  
        }  
  
        return toWrite;  
    }  
  
    // 从缓冲区读取数据  
    size_t read(char* data, size_t len) {  
        size_t toRead = std::min(len, size());  
  
        for (size_t i = 0; i < toRead; ++i) {  
            data[i] = buffer_[tail_];  
            tail_ = (tail_ + 1) % size_;  
        }  
  
        full_ = false;  
  
        return toRead;  
    }  
};
```

```

    }

    // 缓冲区是否为空
    bool empty() const {
        return (!full_ && (head_ == tail_));
    }

    // 缓冲区是否已满
    bool full() const {
        return full_;
    }

    // 缓冲区容量
    size_t capacity() const {
        return size_;
    }

    // 缓冲区当前数据大小
    size_t size() const {
        size_t size = size_;
        if (!full_) {
            if (head_ >= tail_) {
                size = head_ - tail_;
            } else {
                size = size_ + head_ - tail_;
            }
        }
        return size;
    }

private:
    char* buffer_;
    size_t size_;
    MemoryPool& pool_;
    size_t head_;
    size_t tail_;
    bool full_;
};

```

Connection 类

```

cpp
复制代码
#include <string>
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>

// Connection 类，管理每个用户的连接
class Connection {
public:

```

```

Connection(int fd, MemoryPool& pool)
    : fd_(fd), readBuffer_(1024, pool), writeBuffer_(1024, pool) {}

// 发送消息
void send(const std::string& message) {
    uint32_t length = htonl(message.size());
    writeBuffer_.write(reinterpret_cast<const char*>(&length), sizeof(length));
    writeBuffer_.write(message.c_str(), message.size());
    flush();
}

// 刷新缓冲区, 发送数据
void flush() {
    while (!writeBuffer_.empty()) {
        char data[1024];
        size_t len = writeBuffer_.read(data, sizeof(data));
        ::send(fd_, data, len, 0);
    }
}

// 接收数据
void receive() {
    char data[1024];
    ssize_t len = ::recv(fd_, data, sizeof(data), 0);
    if (len > 0) {
        readBuffer_.write(data, len);
        processMessages();
    }
}

private:
// 处理接收到的消息
void processMessages() {
    while (readBuffer_.size() >= sizeof(uint32_t)) {
        uint32_t length;
        readBuffer_.read(reinterpret_cast<char*>(&length), sizeof(length));
        length = ntohl(length);

        if (readBuffer_.size() < length) {
            readBuffer_.write(reinterpret_cast<const char*>(&length),
sizeof(length));
            break;
        }

        char message[1024];
        readBuffer_.read(message, length);
        std::string msg(message, length);
        std::cout << "Received: " << msg << std::endl;
    }
}

int fd_;

```

```
Buffer readBuffer_;
Buffer writeBuffer_;
};
```

详细解释

MemoryPool 类

1. **构造函数**：预先分配固定大小的内存块。
2. **allocate()**：分配内存块，从空闲块栈中取出一个块。
3. **deallocate()**：回收内存块，将块放回空闲块栈中。

Buffer 类

1. **构造函数**：从内存池中分配内存块，并初始化缓冲区指针。
2. **write()**：向缓冲区写入数据，如果缓冲区满，则返回实际写入的数据大小。
3. **read()**：从缓冲区读取数据，如果缓冲区为空，则返回实际读取的数据大小。
4. **empty()** 和 **full()**：判断缓冲区是否为空或已满。
5. **capacity()** 和 **size()**：获取缓冲区容量和当前数据大小。

Connection 类

1. **构造函数**：初始化连接，分配读写缓冲区。
2. **send()**：发送消息，首先写入消息长度，然后写入消息内容。
3. **flush()**：刷新写缓冲区，发送数据。
4. **receive()**：接收数据并写入读缓冲区。
5. **processMessages()**：处理接收到的消息，根据消息长度来区分消息边界。

总结

这种设计使用内存池来减少频繁的内存分配和释放，提高了系统的性能和稳定性。通过在消息前添加长度信息来区分不同的消息边界，有效解决了 TCP 粘包和拆包问题。通过循环缓冲区实现高效的读写操作，避免多次拷贝和内存开辟和销毁。

40

我的epoll模式是et的 也就是只会通知一次 这时候缓冲区的读取和写入策略应该是什么样的 比如有数据读的时候 在epoll通知可读的时候 是不是要一次从sockfd里面全部把数据读完 然后存到自定义缓冲区呢 如果不是 那应该怎么做才对呢

ChatGPT

在使用边缘触发（ET）模式的 `epoll` 时，由于 `epoll` 只会在状态变化时通知一次，因此需要在收到可读或可写事件时，尽可能多地读取或写入数据，以确保不会遗漏数据。这种策略被称为“批量处理”策略。

读取策略

在接收到 `EPOLLIN` 事件时，应该循环读取数据直到没有更多的数据可读（即 `read` 返回 `-1` 且 `errno` 为 `EAGAIN`）。这样可以确保将所有数据读取到自定义缓冲区中。

写入策略

在接收到 `EPOLLOUT` 事件时，应该循环写入数据直到所有数据都写入完毕或缓冲区满（即 `write` 返回 `-1` 且 `errno` 为 `EAGAIN`）。这样可以确保尽可能多地发送数据。

示例代码

以下是一个示例，展示如何在边缘触发模式下处理读写操作：

Connection 类

```
cpp
复制代码
#include <string>
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>

// 设置文件描述符为非阻塞模式
void setNonBlocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        throw std::runtime_error("Failed to get file flags");
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        throw std::runtime_error("Failed to set non-blocking flag");
    }
}

// Connection 类, 管理每个用户的连接
class Connection {
public:
    Connection(int fd, MemoryPool& pool)
        : fd_(fd), readBuffer_(1024, pool), writeBuffer_(1024, pool) {
        setNonBlocking(fd_);
    }

    // 发送消息
    void send(const std::string& message) {
        uint32_t length = htonl(message.size());
        writeBuffer_.write(reinterpret_cast<const char*>(&length), sizeof(length));
        writeBuffer_.write(message.c_str(), message.size());
        flush();
    }
}
```

```

// 刷新缓冲区, 发送数据
void flush() {
    while (!writeBuffer_.empty()) {
        char data[1024];
        size_t len = writeBuffer_.read(data, sizeof(data));
        ssize_t sent = ::send(fd_, data, len, 0);
        if (sent == -1) {
            if (errno == EAGAIN) {
                // 写缓冲区已满, 等待下次可写事件
                writeBuffer_.write(data, len); // 将未发送的数据写回缓冲区
                break;
            } else {
                close(fd_);
                throw std::runtime_error("Send failed");
            }
        }
    }
}

```

```

// 接收数据
void receive() {
    char data[1024];
    while (true) {
        ssize_t len = ::recv(fd_, data, sizeof(data), 0);
        if (len > 0) {
            readBuffer_.write(data, len);
            processMessages();
        } else if (len == -1) {
            if (errno == EAGAIN) {
                // 没有更多数据可读
                break;
            } else {
                close(fd_);
                throw std::runtime_error("Receive failed");
            }
        } else {
            // 对方关闭连接
            close(fd_);
            break;
        }
    }
}

```

private:

```

// 处理接收到的消息
void processMessages() {
    while (readBuffer_.size() >= sizeof(uint32_t)) {
        uint32_t length;
        readBuffer_.read(reinterpret_cast<char*>(&length), sizeof(length));
        length = ntohl(length);
    }
}

```



```

        if (readBuffer_.size() < length) {
            readBuffer_.write(reinterpret_cast<const char*>(&length),
sizeof(length));
            break;
        }

        char message[1024];
        readBuffer_.read(message, length);
        std::string msg(message, length);
        std::cout << "Received: " << msg << std::endl;
    }
}

int fd_;
Buffer readBuffer_;
Buffer writeBuffer_;
};

// 示例使用
int main() {
    // 内存池大小和块大小的设定
    MemoryPool pool(1024, 10);

    // 创建服务器 socket
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket");
        return 1;
    }

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenfd, (sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("bind");
        close(listenfd);
        return 1;
    }

    if (listen(listenfd, 10) == -1) {
        perror("listen");
        close(listenfd);
        return 1;
    }

    // 设置为非阻塞模式
    setNonBlocking(listenfd);

    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {

```

```

        perror("epoll_create1");
        close(listenfd);
        return 1;
    }

    epoll_event event;
    event.data.fd = listenfd;
    event.events = EPOLLIN | EPOLLET;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listenfd, &event) == -1) {
        perror("epoll_ctl");
        close(listenfd);
        close(epoll_fd);
        return 1;
    }

    std::vector<epoll_event> events(10);
    while (true) {
        int n = epoll_wait(epoll_fd, events.data(), events.size(), -1);
        if (n == -1) {
            perror("epoll_wait");
            break;
        }

        for (int i = 0; i < n; ++i) {
            if (events[i].data.fd == listenfd) {
                while (true) {
                    int connfd = accept(listenfd, nullptr, nullptr);
                    if (connfd == -1) {
                        if (errno == EAGAIN || errno == EWOULDBLOCK) {
                            break;
                        } else {
                            perror("accept");
                            break;
                        }
                    }

                    setNonBlocking(connfd);

                    epoll_event connEvent;
                    connEvent.data.ptr = new Connection(connfd, pool);
                    connEvent.events = EPOLLIN | EPOLLET | EPOLLOUT;
                    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, connfd, &connEvent) == -1) {
                        perror("epoll_ctl");
                        delete static_cast<Connection*>(connEvent.data.ptr);
                        close(connfd);
                    }
                }
            } else {
                Connection* conn = static_cast<Connection*>(events[i].data.ptr);
                if (events[i].events & EPOLLIN) {
                    conn->receive();
                }
            }
        }
    }
}

```

```

        if (events[i].events & EPOLLOUT) {
            conn->flush();
        }
    }
}

close(listenfd);
close(epoll_fd);
return 0;
}

```

详细解释和注释

设置文件描述符为非阻塞模式

```

cpp
复制代码
void setNonBlocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        throw std::runtime_error("Failed to get file flags");
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        throw std::runtime_error("Failed to set non-blocking flag");
    }
}

```

Connection 类

- **构造函数**：初始化连接，设置文件描述符为非阻塞模式。

```

cpp
复制代码
Connection(int fd, MemoryPool& pool)
    : fd_(fd), readBuffer_(1024, pool), writeBuffer_(1024, pool) {
    setNonBlocking(fd_);
}

```

- **send() 方法**：发送消息，首先写入消息长度，然后写入消息内容，并调用 `flush` 方法发送数据。

```

cpp
复制代码
void send(const std::string& message) {
    uint32_t length = htonl(message.size());
    writeBuffer_.write(reinterpret_cast<const char*>(&length), sizeof(length));
    writeBuffer_.write(message.c_str(), message.size());
    flush();
}

```

- **flush()** 方法：从写缓冲区读取数据并发送，直到写缓冲区为空或写操作返回 `EAGAIN`。

cpp

复制代码

```
void flush() {
    while (!writeBuffer_.empty()) {
        char data[1024];
        size_t len = writeBuffer_.read(data, sizeof(data));
        ssize_t sent = ::send(fd_, data, len, 0);
        if (sent == -1) {
            if (errno == EAGAIN) {
                writeBuffer_.write(data, len); // 将未发送的数据写回缓冲区
                break;
            } else {
                close(fd_);
                throw std::runtime_error("Send failed");
            }
        }
    }
}
```

- **receive()** 方法：从套接字读取数据并写入读缓冲区，直到没有更多数据可读（`read` 返回 `-1` 且 `errno` 为 `EAGAIN`）。

cpp

复制代码

```
void receive() {
    char data[1024];
    while (true) {
        ssize_t len = ::recv(fd_, data, sizeof(data), 0);
        if (len > 0) {
            readBuffer_.write(data, len);
            processMessages();
        } else if (len == -1) {
            if (errno == EAGAIN) {
                break;
            } else {
                close(fd_);
                throw std::runtime_error("Receive failed");
            }
        } else {
            close(fd_);
            break;
        }
    }
}
```

- **processMessages()** 方法：处理接收到的消息，根据消息长度来区分消息边界。

cpp

复制代码

```
void processMessages() {
    while (readBuffer_.size() >= sizeof(uint32_t)) {
        uint32_t length;
        readBuffer_.read(reinterpret_cast<char*>(&length), sizeof(length));
        length = ntohl(length);

        if (readBuffer_.size() < length) {
            readBuffer_.write(reinterpret_cast<const char*>(&length), sizeof(length));
            break;
        }

        char message[1024];
        readBuffer_.read(message, length);
        std::string msg(message, length);
        std::cout << "Received: " << msg << std::endl;
    }
}
```

主函数

- 设置服务器套接字为非阻塞模式。

cpp

复制代码

```
int listenfd = socket(AF_INET, SOCK_STREAM, 0);
if (listenfd == -1) {
    perror("socket");
    return 1;
}

sockaddr_in serverAddr;
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(8080);
serverAddr.sin_addr.s_addr = INADDR_ANY;

if (bind(listenfd, (sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
    perror("bind");
    close(listenfd);
    return 1;
}

if (listen(listenfd, 10) == -1) {
    perror("listen");
    close(listenfd);
    return 1;
}

setNonBlocking(listenfd);
```

- 创建 **epoll** 实例并将监听套接字添加到 **epoll** 中。

cpp

复制代码

```
int epoll_fd = epoll_create1(0);
if (epoll_fd == -1) {
    perror("epoll_create1");
    close(listenfd);
    return 1;
}

epoll_event event;
event.data.fd = listenfd;
event.events = EPOLLIN | EPOLLET;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listenfd, &event) == -1) {
    perror("epoll_ctl");
    close(listenfd);
    close(epoll_fd);
    return 1;
}
```

- 在事件循环中处理事件。

cpp

复制代码

```
std::vector<epoll_event> events(10);
while (true) {
    int n = epoll_wait(epoll_fd, events.data(), events.size(), -1);
    if (n == -1) {
        perror("epoll_wait");
        break;
    }

    for (int i = 0; i < n; ++i) {
        if (events[i].data.fd == listenfd) {
            while (true) {
                int connfd = accept(listenfd, nullptr, nullptr);
                if (connfd == -1) {
                    if (errno == EAGAIN || errno == EWOULDBLOCK) {
                        break;
                    } else {
                        perror("accept");
                        break;
                    }
                }
            }

            setNonBlocking(connfd);

            epoll_event connEvent;
            connEvent.data.ptr = new Connection(connfd, pool);
            connEvent.events = EPOLLIN | EPOLLET | EPOLLOUT;
```

```

        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, connfd, &connEvent) == -1) {
            perror("epoll_ctl");
            delete static_cast<Connection*>(connEvent.data.ptr);
            close(connfd);
        }
    }
} else {
    Connection* conn = static_cast<Connection*>(events[i].data.ptr);
    if (events[i].events & EPOLLIN) {
        conn->receive();
    }
    if (events[i].events & EPOLLOUT) {
        conn->flush();
    }
}
}
}

close(listenfd);
close(epoll_fd);
return 0;

```

这种设计通过使用非阻塞 I/O 和边缘触发模式，确保在有数据可读或可写时尽可能多地处理数据，避免了多次通知带来的性能开销。通过 `MemoryPool` 和 `Buffer` 的结合使用，减少了内存分配和释放的频率，提高了系统的效率。

异步IO

这里怎么结合异步io来提高性能呢 不能使用boost库等 只能使用aio或者c++的异步接口

ChatGPT

为了结合异步 I/O 提高性能，我们可以使用 Linux 提供的 `aio`（异步 I/O）库来处理 I/O 操作。异步 I/O 允许我们在不阻塞的情况下处理读写操作，从而提高系统的并发性能。以下是如何在现有的设计中结合异步 I/O 的详细步骤和代码示例。

使用 Linux AIO 库

Linux 提供了 `libaio` 库来支持异步 I/O 操作，我们将使用该库来处理读写操作。

安装 libaio

在使用 `libaio` 之前，需要确保其已安装在系统中。可以通过以下命令安装：

```

sh
复制代码
sudo apt-get install libaio-dev

```

异步 I/O 设计思路

1. 初始化异步 I/O：初始化异步 I/O 上下文。
2. 提交异步 I/O 请求：提交读写请求，设置回调函数。
3. 处理完成事件：在回调函数中处理完成的 I/O 请求。

代码实现

初始化异步 I/O 上下文

```
cpp
复制代码
#include <libaio.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
#include <cstring>
#include <vector>

const int MAX_EVENTS = 128;

class AIOContext {
public:
    AIOContext() {
        memset(&ctx, 0, sizeof(ctx));
        if (io_setup(MAX_EVENTS, &ctx) < 0) {
            perror("io_setup");
            throw std::runtime_error("Failed to setup aio context");
        }
    }

    ~AIOContext() {
        io_destroy(ctx);
    }

    io_context_t getContext() {
        return ctx;
    }

private:
    io_context_t ctx;
};
```

Connection 类的修改

在 Connection 类中使用异步 I/O 处理读写操作。

```
cpp
复制代码
#include <libaio.h>
```



```

#include <arpa/inet.h>
#include <cstring>
#include <unistd.h>
#include <vector>
#include <stdexcept>

class Connection {
public:
    Connection(int fd, AIOContext& aioCtx)
        : fd_(fd), aioCtx_(aioCtx), readInProgress_(false), writeInProgress_(false) {
        setNonBlocking(fd_);
    }

    ~Connection() {
        close(fd_);
    }

    void startRead() {
        if (readInProgress_) return;

        readInProgress_ = true;
        iocb cb;
        iocb* cbs[1];
        char* buffer = new char[BUFFER_SIZE];
        memset(&cb, 0, sizeof(cb));
        io_prep_pread(&cb, fd_, buffer, BUFFER_SIZE, 0);
        cb.data = buffer;

        cbs[0] = &cb;
        if (io_submit(aioCtx_.getContext(), 1, cbs) < 0) {
            perror("io_submit");
            throw std::runtime_error("Failed to submit read request");
        }
    }

    void startWrite(const std::string& data) {
        if (writeInProgress_) return;

        writeInProgress_ = true;
        iocb cb;
        iocb* cbs[1];
        char* buffer = new char[data.size()];
        memcpy(buffer, data.c_str(), data.size());
        memset(&cb, 0, sizeof(cb));
        io_prep_pwrite(&cb, fd_, buffer, data.size(), 0);
        cb.data = buffer;

        cbs[0] = &cb;
        if (io_submit(aioCtx_.getContext(), 1, cbs) < 0) {
            perror("io_submit");
            throw std::runtime_error("Failed to submit write request");
        }
    }

```

```

    }

    void handleReadCompletion(io_event* event) {
        char* buffer = static_cast<char*>(event->data);
        ssize_t bytesRead = event->res;
        if (bytesRead > 0) {
            std::string message(buffer, bytesRead);
            std::cout << "Received: " << message << std::endl;
            // Process the message here
        }
        delete[] buffer;
        readInProgress_ = false;
    }

    void handleWriteCompletion(io_event* event) {
        char* buffer = static_cast<char*>(event->data);
        delete[] buffer;
        writeInProgress_ = false;
    }

private:
    void setNonBlocking(int fd) {
        int flags = fcntl(fd, F_GETFL, 0);
        if (flags == -1) {
            throw std::runtime_error("Failed to get file flags");
        }
        if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
            throw std::runtime_error("Failed to set non-blocking flag");
        }
    }

    int fd_;
    AIOContext& aioCtx_;
    bool readInProgress_;
    bool writeInProgress_;
    static const int BUFFER_SIZE = 1024;
};

```

主循环和事件处理

在主循环中使用 `epoll` 等待事件，然后使用 `io_getevents` 获取异步 I/O 事件并处理。

```

cpp
复制代码
#include <sys/epoll.h>
#include <unistd.h>
#include <vector>
#include <iostream>

void eventLoop(int listenfd, AIOContext& aioCtx) {
    int epoll_fd = epoll_create1(0);

```

```

if (epoll_fd == -1) {
    perror("epoll_create1");
    close(listenfd);
    return;
}

epoll_event event;
event.data.fd = listenfd;
event.events = EPOLLIN | EPOLLET;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listenfd, &event) == -1) {
    perror("epoll_ctl");
    close(listenfd);
    close(epoll_fd);
    return;
}

std::vector<epoll_event> events(MAX_EVENTS);

while (true) {
    int n = epoll_wait(epoll_fd, events.data(), events.size(), -1);
    if (n == -1) {
        perror("epoll_wait");
        break;
    }

    for (int i = 0; i < n; ++i) {
        if (events[i].data.fd == listenfd) {
            while (true) {
                int connfd = accept(listenfd, nullptr, nullptr);
                if (connfd == -1) {
                    if (errno == EAGAIN || errno == EWOULDBLOCK) {
                        break;
                    } else {
                        perror("accept");
                        break;
                    }
                }
            }

            setNonBlocking(connfd);

            Connection* conn = new Connection(connfd, aioCtx);
            conn->startRead();

            epoll_event connEvent;
            connEvent.data.ptr = conn;
            connEvent.events = EPOLLIN | EPOLLET;
            if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, connfd, &connEvent) == -1) {
                perror("epoll_ctl");
                delete conn;
                close(connfd);
            }
        }
    }
}

```

```

        } else {
            Connection* conn = static_cast<Connection*>(events[i].data.ptr);
            io_event ioEvents[MAX_EVENTS];
            int numEvents = io_getevents(aioCtx.getContext(), 1, MAX_EVENTS,
ioEvents, nullptr);
            for (int j = 0; j < numEvents; ++j) {
                if (ioEvents[j].res >= 0) {
                    if (events[i].events & EPOLLIN) {
                        conn->handleReadCompletion(&ioEvents[j]);
                    }
                    if (events[i].events & EPOLLOUT) {
                        conn->handleWriteCompletion(&ioEvents[j]);
                    }
                } else {
                    perror("io_getevents");
                    delete conn;
                    close(conn->fd_);
                }
            }
        }
    }
}

close(listenfd);
close(epoll_fd);
}

int main() {
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket");
        return 1;
    }

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenfd, (sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("bind");
        close(listenfd);
        return 1;
    }

    if (listen(listenfd, 10) == -1) {
        perror("listen");
        close(listenfd);
        return 1;
    }

    setNonBlocking(listenfd);

```

```

    try {
        AIOContext aioCtx;
        eventLoop(listenfd, aioCtx);
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }

    return 0;
}

```

详细说明

1. **AIOContext** 类：管理异步 I/O 上下文。
2. **Connection** 类
 - ：管理每个用户的连接，处理异步读写操作。
 - **startRead** 和 **startWrite**：提交异步 I/O 请求。
 - **handleReadCompletion** 和 **handleWriteCompletion**：处理异步 I/O 完成事件。
3. 主循环：使用 `epoll` 等待事件，并使用 `io_getevents` 获取异步 I/O 事件。

这种设计结合了异步 I/O 和 `epoll`，在 `epoll` 事件触发后，使用异步 I/O 提交读写请求，并在请求完成后处理相应的事件，从而提高系统的并发性能。

4o

代码给出注释啊 详细讲讲异步io接口怎么用的 代码加上清楚的注释 并给出解释啊

ChatGPT

好的，我将详细解释异步 I/O 接口的使用，并在代码中添加详细注释。

异步 I/O 介绍

在 Linux 中，异步 I/O (AIO) 允许应用程序在不阻塞的情况下发起 I/O 操作，并在操作完成时通过回调或事件通知应用程序。这种机制可以极大地提高 I/O 密集型应用程序的性能。

AIO 基本步骤

1. 初始化异步 I/O 上下文：使用 `io_setup` 初始化异步 I/O 上下文。
2. 准备 I/O 控制块 (iocb)：使用 `io_prep_pread` 和 `io_prep_pwrite` 函数准备读写操作。
3. 提交 I/O 请求：使用 `io_submit` 提交 I/O 请求。
4. 等待 I/O 完成：使用 `io_getevents` 等待并处理 I/O 完成事件。
5. 销毁异步 I/O 上下文：使用 `io_destroy` 销毁异步 I/O 上下文。

代码实现

AIOContext 类

用于管理异步 I/O 上下文。

```
cpp
复制代码
#include <libaio.h>
#include <stdexcept>
#include <cstring>

// AIOContext 类, 用于管理异步 I/O 上下文
class AIOContext {
public:
    AIOContext() {
        memset(&ctx, 0, sizeof(ctx));
        if (io_setup(MAX_EVENTS, &ctx) < 0) {
            perror("io_setup");
            throw std::runtime_error("Failed to setup aio context");
        }
    }

    ~AIOContext() {
        io_destroy(ctx);
    }

    io_context_t getContext() {
        return ctx;
    }

private:
    io_context_t ctx;
    static const int MAX_EVENTS = 128; // 最大事件数
};
```

Connection 类

管理每个用户的连接, 处理异步读写操作。

```
cpp
复制代码
#include <iostream>
#include <unistd.h>
#include <fcntl.h>
#include <cstring>
#include <arpa/inet.h>

// 设置文件描述符为非阻塞模式
void setNonBlocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        throw std::runtime_error("Failed to get file flags");
    }
}
```

```

    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        throw std::runtime_error("Failed to set non-blocking flag");
    }
}

// Connection 类, 管理每个用户的连接
class Connection {
public:
    Connection(int fd, AIOContext& aioCtx)
        : fd_(fd), aioCtx_(aioCtx), readInProgress_(false), writeInProgress_(false) {
        setNonBlocking(fd_);
    }

    ~Connection() {
        close(fd_);
    }

    // 开始异步读取数据
    void startRead() {
        if (readInProgress_) return;

        readInProgress_ = true;
        iocb cb;
        iocb* cbs[1];
        char* buffer = new char[BUFFER_SIZE];
        memset(&cb, 0, sizeof(cb));
        io_prep_pread(&cb, fd_, buffer, BUFFER_SIZE, 0);
        cb.data = buffer;

        cbs[0] = &cb;
        if (io_submit(aioCtx_.getContext(), 1, cbs) < 0) {
            perror("io_submit");
            throw std::runtime_error("Failed to submit read request");
        }
    }

    // 开始异步写入数据
    void startWrite(const std::string& data) {
        if (writeInProgress_) return;

        writeInProgress_ = true;
        iocb cb;
        iocb* cbs[1];
        char* buffer = new char[data.size()];
        memcpy(buffer, data.c_str(), data.size());
        memset(&cb, 0, sizeof(cb));
        io_prep_pwrite(&cb, fd_, buffer, data.size(), 0);
        cb.data = buffer;

        cbs[0] = &cb;
        if (io_submit(aioCtx_.getContext(), 1, cbs) < 0) {

```

```

        perror("io_submit");
        throw std::runtime_error("Failed to submit write request");
    }
}

// 处理读操作完成事件
void handleReadCompletion(io_event* event) {
    char* buffer = static_cast<char*>(event->data);
    ssize_t bytesRead = event->res;
    if (bytesRead > 0) {
        std::string message(buffer, bytesRead);
        std::cout << "Received: " << message << std::endl;
        // 在此处理接收到的消息
    }
    delete[] buffer;
    readInProgress_ = false;
    startRead(); // 再次启动读操作
}

// 处理写操作完成事件
void handleWriteCompletion(io_event* event) {
    char* buffer = static_cast<char*>(event->data);
    delete[] buffer;
    writeInProgress_ = false;
}

private:
    int fd_;
    AIOContext& aioCtx_;
    bool readInProgress_;
    bool writeInProgress_;
    static const int BUFFER_SIZE = 1024;
};

```

主循环和事件处理

在主循环中使用 `epoll` 等待事件，然后使用 `io_getevents` 获取异步 I/O 事件并处理。

```

cpp
复制代码
#include <sys/epoll.h>
#include <unistd.h>
#include <vector>
#include <iostream>

void eventLoop(int listenfd, AIOContext& aioCtx) {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        close(listenfd);
        return;
    }
}

```



```

}

epoll_event event;
event.data.fd = listenfd;
event.events = EPOLLIN | EPOLLET;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listenfd, &event) == -1) {
    perror("epoll_ctl");
    close(listenfd);
    close(epoll_fd);
    return;
}

std::vector<epoll_event> events(AIOContext::MAX_EVENTS);

while (true) {
    int n = epoll_wait(epoll_fd, events.data(), events.size(), -1);
    if (n == -1) {
        perror("epoll_wait");
        break;
    }

    for (int i = 0; i < n; ++i) {
        if (events[i].data.fd == listenfd) {
            while (true) {
                int connfd = accept(listenfd, nullptr, nullptr);
                if (connfd == -1) {
                    if (errno == EAGAIN || errno == EWOULDBLOCK) {
                        break;
                    } else {
                        perror("accept");
                        break;
                    }
                }
            }

            setNonBlocking(connfd);

            Connection* conn = new Connection(connfd, aioCtx);
            conn->startRead();

            epoll_event connEvent;
            connEvent.data.ptr = conn;
            connEvent.events = EPOLLIN | EPOLLET;
            if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, connfd, &connEvent) == -1) {
                perror("epoll_ctl");
                delete conn;
                close(connfd);
            }
        }
        else {
            Connection* conn = static_cast<Connection*>(events[i].data.ptr);
            io_event ioEvents[AIOContext::MAX_EVENTS];

```

```

        int numEvents = io_getevents(aioCtx.getContext(), 1,
AIOContext::MAX_EVENTS, ioEvents, nullptr);
        for (int j = 0; j < numEvents; ++j) {
            if (ioEvents[j].res >= 0) {
                if (events[i].events & EPOLLIN) {
                    conn->handleReadCompletion(&ioEvents[j]);
                }
                if (events[i].events & EPOLLOUT) {
                    conn->handleWriteCompletion(&ioEvents[j]);
                }
            } else {
                perror("io_getevents");
                delete conn;
                close(conn->fd_);
            }
        }
    }
}

close(listenfd);
close(epoll_fd);
}

int main() {
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket");
        return 1;
    }

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenfd, (sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("bind");
        close(listenfd);
        return 1;
    }

    if (listen(listenfd, 10) == -1) {
        perror("listen");
        close(listenfd);
        return 1;
    }

    setNonBlocking(listenfd);

    try {
        AIOContext aioCtx;

```

```

        eventLoop(listenfd, aioCtx);
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << std::endl;
    }

    return 0;
}

```

详细说明和注释

1. AIOContext 类

：管理异步 I/O 上下文，包括初始化和销毁。

- `io_setup`：初始化异步 I/O 上下文。
- `io_destroy`：销毁异步 I/O 上下文。

```

cpp
复制代码
class AIOContext {
public:
    AIOContext() {
        memset(&ctx, 0, sizeof(ctx));
        if (io_setup(MAX_EVENTS, &ctx) < 0) {
            perror("io_setup");
            throw std::runtime_error("Failed to setup aio context");
        }
    }

    ~AIOContext() {
        io_destroy(ctx);
    }

    io_context_t getContext() {
        return ctx;
    }

private:
    io_context_t ctx;
    static const int MAX_EVENTS = 128; // 最大事件数
};

```

1. Connection 类

：管理每个用户的连接，处理异步读写操作。

- `startRead` 和 `startWrite`：准备并提交异步读写请求。
- `handleReadCompletion` 和 `handleWriteCompletion`：处理异步读写完成事件。

```

cpp
复制代码
class Connection {

```

```

public:
    Connection(int fd, AIOContext& aioCtx)
        : fd_(fd), aioCtx_(aioCtx), readInProgress_(false), writeInProgress_(false) {
        setNonBlocking(fd_);
    }

    ~Connection() {
        close(fd_);
    }

    // 开始异步读取数据
    void startRead() {
        if (readInProgress_) return;

        readInProgress_ = true;
        iocb cb;
        iocb* cbs[1];
        char* buffer = new char[BUFFER_SIZE];
        memset(&cb, 0, sizeof(cb));
        io_prep_pread(&cb, fd_, buffer, BUFFER_SIZE, 0);
        cb.data = buffer;

        cbs[0] = &cb;
        if (io_submit(aioCtx_.getContext(), 1, cbs) < 0) {
            perror("io_submit");
            throw std::runtime_error("Failed to submit read request");
        }
    }

    // 开始异步写入数据
    void startWrite(const std::string& data) {
        if (writeInProgress_) return;

        writeInProgress_ = true;
        iocb cb;
        iocb* cbs[1];
        char* buffer = new char[data.size()];
        memcpy(buffer, data.c_str(), data.size());
        memset(&cb, 0, sizeof(cb));
        io_prep_pwrite(&cb, fd_, buffer, data.size(), 0);
        cb.data = buffer;

        cbs[0] = &cb;
        if (io_submit(aioCtx_.getContext(), 1, cbs) < 0) {
            perror("io_submit");
            throw std::runtime_error("Failed to submit write request");
        }
    }

    // 处理读操作完成事件
    void handleReadCompletion(io_event* event) {
        char* buffer = static_cast<char*>(event->data);
    }

```

```

        ssize_t bytesRead = event->res;
        if (bytesRead > 0) {
            std::string message(buffer, bytesRead);
            std::cout << "Received: " << message << std::endl;
            // 在此处理接收到的消息
        }
        delete[] buffer;
        readInProgress_ = false;
        startRead(); // 再次启动读操作
    }

    // 处理写操作完成事件
    void handleWriteCompletion(io_event* event) {
        char* buffer = static_cast<char*>(event->data);
        delete[] buffer;
        writeInProgress_ = false;
    }

private:
    int fd_;
    AIOContext& aioCtx_;
    bool readInProgress_;
    bool writeInProgress_;
    static const int BUFFER_SIZE = 1024;
};

```

1. **主循环**: 使用 `epoll` 等待事件, 并使用 `io_getevents` 获取异步 I/O 事件。

```

cpp
复制代码
void eventLoop(int listenfd, AIOContext& aioCtx) {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        close(listenfd);
        return;
    }

    epoll_event event;
    event.data.fd = listenfd;
    event.events = EPOLLIN | EPOLLET;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listenfd, &event) == -1) {
        perror("epoll_ctl");
        close(listenfd);
        close(epoll_fd);
        return;
    }

    std::vector<epoll_event> events(AIOContext::MAX_EVENTS);

    while (true) {

```

```

int n = epoll_wait(epoll_fd, events.data(), events.size(), -1);
if (n == -1) {
    perror("epoll_wait");
    break;
}

for (int i = 0; i < n; ++i) {
    if (events[i].data.fd == listenfd) {
        while (true) {
            int connfd = accept(listenfd, nullptr, nullptr);
            if (connfd == -1) {
                if (errno == EAGAIN || errno == EWOULDBLOCK) {
                    break;
                } else {
                    perror("accept");
                    break;
                }
            }

            setNonBlocking(connfd);

            Connection* conn = new Connection(connfd, aioCtx);
            conn->startRead();

            epoll_event connEvent;
            connEvent.data.ptr = conn;
            connEvent.events = EPOLLIN | EPOLLET;
            if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, connfd, &connEvent) == -1) {
                perror("epoll_ctl");
                delete conn;
                close(connfd);
            }
        }
    } else {
        Connection* conn = static_cast<Connection*>(events[i].data.ptr);
        io_event ioEvents[AIOContext::MAX_EVENTS];
        int numEvents = io_getevents(aioCtx.getContext(), 1,
AIOContext::MAX_EVENTS, ioEvents, nullptr);
        for (int j = 0; j < numEvents; ++j) {
            if (ioEvents[j].res >= 0) {
                if (events[i].events & EPOLLIN) {
                    conn->handleReadCompletion(&ioEvents[j]);
                }
                if (events[i].events & EPOLLOUT) {
                    conn->handleWriteCompletion(&ioEvents[j]);
                }
            } else {
                perror("io_getevents");
                delete conn;
                close(conn->fd_);
            }
        }
    }
}

```

```

    }
}

close(listenfd);
close(epoll_fd);
}

```

总结

1. 初始化异步 I/O 上下文：使用 `io_setup` 初始化异步 I/O 上下文。
2. 准备并提交 I/O 请求：使用 `io_prep_read` 和 `io_prep_pwrite` 函数准备读写操作，并使用 `io_submit` 提交请求。
3. 处理完成事件：在 `handleReadCompletion` 和 `handleWriteCompletion` 方法中处理异步 I/O 完成事件。
4. 事件循环：使用 `epoll` 等待事件，并使用 `io_getevents` 获取异步 I/O 事件。

通过这种设计，结合异步 I/O 和 `epoll`，可以在高并发场景下提高系统性能，并避免阻塞操作。

1. 讲一下 send close recv write read（还有涉及网络编程的接口等）这些接口的区别 2. 在这个 tcp server 项目设计里面 原生的 send recv close 接口是线程安全的吗 我现在会遇到这样的情况 客户端拿到 connection 对象后 因为 connection 对象对应一个连接 客户端拿到这个链接后 可能在应用层做其他的应用 那么可能应用层也有多线程场景 会对同一个 connection 进行操作 那么我该怎么保证里面的 send close 等接口是线程安全的呢 我该怎么管理这些 connection 呢？

ChatGPT

1. 网络编程接口的区别

在网络编程中，`send`，`recv`，`close`，`write`，`read` 等接口具有不同的用途和行为。以下是这些接口的详细区别：

`send` 和 `recv`

- **用途**：用于在 TCP 连接上发送和接收数据。
- **特点**：特定于网络套接字，通常与 `sendto` 和 `recvfrom` 配合使用。

C

复制代码

```

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);

```

- `send`：向套接字发送数据，可以指定发送的标志位（如 `MSG_DONTWAIT`）。
- `recv`：从套接字接收数据，可以指定接收的标志位（如 `MSG_DONTWAIT`）。

write 和 read

- **用途**：用于在文件描述符上写入和读取数据，适用于所有类型的文件描述符（包括网络套接字、文件、管道等）。
- **特点**：通用的 I/O 操作函数。

C

复制代码

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

- `write`：向文件描述符写入数据。
- `read`：从文件描述符读取数据。

close

- **用途**：关闭文件描述符，释放相关资源。
- **特点**：通用的文件描述符关闭函数。

C

复制代码

```
int close(int fd);
```

- `close`：关闭文件描述符，不再使用该描述符。

2. 线程安全性和连接管理

原生接口的线程安全性

- **线程安全性**：`send`, `recv`, `write`, `read` 和 `close` 等接口在设计上并不是线程安全的。当多个线程同时对同一个套接字进行操作时，可能会导致数据混乱、资源竞争等问题。

解决方法

为了确保线程安全性，可以采取以下措施：

1. **使用互斥锁（Mutex）**：在对同一个 `Connection` 对象进行 `send`, `recv`, `close` 等操作时使用互斥锁，确保操作的原子性。
2. **线程安全的队列**：使用线程安全的队列将数据包排队，确保多个线程同时操作时不会出现竞争条件。
3. **连接管理**：维护一个 `Connection` 对象池，用于管理所有的连接对象。

示例代码

以下是一个示例，展示如何实现线程安全的 `Connection` 类，并使用互斥锁保护 `send`, `recv`, `close` 等操作。

C++

复制代码

```
#include <iostream>
```



```

#include <mutex>
#include <vector>
#include <string>
#include <sys/socket.h>
#include <unistd.h>
#include <fcntl.h>
#include <arpa/inet.h>

// 设置文件描述符为非阻塞模式
void setNonBlocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        throw std::runtime_error("Failed to get file flags");
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        throw std::runtime_error("Failed to set non-blocking flag");
    }
}

// 线程安全的 Connection 类
class Connection {
public:
    Connection(int fd) : fd_(fd) {
        setNonBlocking(fd_);
    }

    ~Connection() {
        close(fd_);
    }

    // 发送数据
    void send(const std::string& data) {
        std::lock_guard<std::mutex> lock(mutex_);
        ::send(fd_, data.c_str(), data.size(), 0);
    }

    // 接收数据
    std::string recv() {
        std::lock_guard<std::mutex> lock(mutex_);
        char buffer[1024];
        ssize_t len = ::recv(fd_, buffer, sizeof(buffer), 0);
        if (len > 0) {
            return std::string(buffer, len);
        } else {
            return "";
        }
    }

    // 关闭连接
    void close() {
        std::lock_guard<std::mutex> lock(mutex_);
        if (fd_ != -1) {

```

```

        ::close(fd_);
        fd_ = -1;
    }
}

private:
    int fd_;
    std::mutex mutex_;
};

// 连接管理类
class ConnectionManager {
public:
    void addConnection(int fd) {
        std::lock_guard<std::mutex> lock(mutex_);
        connections_.emplace_back(new Connection(fd));
    }

    void removeConnection(Connection* conn) {
        std::lock_guard<std::mutex> lock(mutex_);
        connections_.erase(std::remove(connections_.begin(), connections_.end(), conn),
connections_.end());
        delete conn;
    }

    std::vector<Connection*> getConnections() {
        std::lock_guard<std::mutex> lock(mutex_);
        return connections_;
    }

private:
    std::vector<Connection*> connections_;
    std::mutex mutex_;
};

// 示例使用
int main() {
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket");
        return 1;
    }

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenfd, (sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("bind");
        close(listenfd);
        return 1;
    }
}

```

```

    }

    if (listen(listenfd, 10) == -1) {
        perror("listen");
        close(listenfd);
        return 1;
    }

    setNonBlocking(listenfd);

    ConnectionManager connManager;

    while (true) {
        int connfd = accept(listenfd, nullptr, nullptr);
        if (connfd == -1) {
            if (errno != EAGAIN && errno != EWOULDBLOCK) {
                perror("accept");
            }
            continue;
        }

        setNonBlocking(connfd);
        connManager.addConnection(connfd);
    }

    close(listenfd);
    return 0;
}

```

解释

1. Connection 类：

- 使用 `std::mutex` 确保 `send`, `recv`, `close` 等操作的线程安全性。
- 通过锁机制保护这些操作，防止多线程同时操作同一个 `Connection` 对象时出现数据竞争问题。

2. ConnectionManager 类：

- 管理所有的连接对象，提供添加和删除连接的方法。
- 使用互斥锁保护对连接对象列表的操作，确保线程安全。

3. 主函数：

- 创建服务器套接字并设置为非阻塞模式。
- 在主循环中接受新的连接，并将其添加到 `ConnectionManager` 中。

通过这种设计，可以确保在多线程环境中对 `Connection` 对象的操作是线程安全的。所有对 `send`, `recv`, `close` 等操作的访问都被互斥锁保护，防止竞争条件和数据损坏。

Wakeup FD 的特点

在多线程网络编程中，`wakeup fd`（或 `eventfd`）是一种用于线程间通知和同步的机制，特别适用于 `epoll` 等多路复用 I/O 模型中。`eventfd` 提供了一种轻量级的事件通知机制，可以被用来唤醒等待 I/O 事件的线程。

特点

1. **轻量级**：`eventfd` 是一种轻量级的事件通知机制，开销较低，适合高性能应用。
2. **简单易用**：API 简单，主要通过 `eventfd` 创建、`read` 和 `write` 操作进行使用。
3. **线程间通信**：可以用于线程间的简单通知机制，避免使用复杂的锁机制。
4. **适用于 `epoll`**：可以与 `epoll` 结合使用，用于唤醒被阻塞的 `epoll_wait` 调用。

`eventfd` API

创建 `eventfd`

```
c
复制代码
#include <sys/eventfd.h>

int efd = eventfd(unsigned int initval, int flags);
```

- `initval`：初始计数值。
- `flags`：可以是 0，或者 `EFD_NONBLOCK`（非阻塞模式），或者 `EFD_SEMAPHORE`（信号量模式）。

读取和写入 `eventfd`

- **读取 `eventfd`**：读取当前计数值。

```
c
复制代码
#include <unistd.h>

uint64_t u;
ssize_t n = read(efd, &u, sizeof(uint64_t));
```

- **写入 `eventfd`**：增加计数值。

```
c
复制代码
uint64_t u = 1;
ssize_t n = write(efd, &u, sizeof(uint64_t));
```

使用示例

以下是一个示例，展示如何使用 `eventfd` 来唤醒被 `epoll` 阻塞的线程：

```
cpp
复制代码
```

```

#include <iostream>
#include <sys/epoll.h>
#include <sys/eventfd.h>
#include <unistd.h>
#include <thread>
#include <vector>

// 设置文件描述符为非阻塞模式
void setNonBlocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        throw std::runtime_error("Failed to get file flags");
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        throw std::runtime_error("Failed to set non-blocking flag");
    }
}

void workerThread(int epoll_fd, int efd) {
    epoll_event events[10];

    while (true) {
        int n = epoll_wait(epoll_fd, events, 10, -1);
        if (n == -1) {
            perror("epoll_wait");
            break;
        }

        for (int i = 0; i < n; ++i) {
            if (events[i].data.fd == efd) {
                uint64_t u;
                read(efd, &u, sizeof(uint64_t)); // 清空 eventfd
                std::cout << "Worker thread woke up" << std::endl;
                // 处理其他事件...
            }
        }
    }
}

int main() {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        return 1;
    }

    int efd = eventfd(0, EFD_NONBLOCK);
    if (efd == -1) {
        perror("eventfd");
        close(epoll_fd);
        return 1;
    }
}

```

```

    epoll_event event;
    event.data.fd = efd;
    event.events = EPOLLIN | EPOLLET;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, efd, &event) == -1) {
        perror("epoll_ctl");
        close(efd);
        close(epoll_fd);
        return 1;
    }

    std::thread worker(workerThread, epoll_fd, efd);

    // 主线程向 eventfd 写入数据，唤醒工作线程
    uint64_t u = 1;
    write(efd, &u, sizeof(uint64_t));

    worker.join();
    close(efd);
    close(epoll_fd);
    return 0;
}

```

解释

1. 创建 **eventfd**：通过 `eventfd(0, EFD_NONBLOCK)` 创建一个非阻塞的 **eventfd**，初始值为 0。
2. 添加到 **epoll**：将 **eventfd** 添加到 **epoll** 实例中，监视 **EPOLLIN** 事件。
3. 工作线程：在 `workerThread` 中，调用 `epoll_wait` 等待事件。当 **eventfd** 有可读事件时，读取计数值并处理事件。
4. 主线程唤醒：主线程向 **eventfd** 写入数据（增加计数值），唤醒工作线程。

线程安全性

eventfd 本身是线程安全的，可以在多个线程中同时进行读写操作。写入 **eventfd** 会增加计数值，读取 **eventfd** 会清空计数值（读取到当前的计数值并将其清零）。

总结

- **特点**：**eventfd** 是一种轻量级的事件通知机制，适用于线程间的简单通知和同步。
- **用法**：通过 **eventfd** 创建事件文件描述符，使用 `read` 和 `write` 进行通知，适用于与 **epoll** 结合使用。
- **线程安全性**：**eventfd** 本身是线程安全的，可以在多线程环境中安全使用。

在多线程网络编程中，使用 **eventfd** 可以有效地唤醒阻塞在 `epoll_wait` 上的线程，从而处理新的 I/O 事件，避免复杂的锁机制，提高系统性能。

自定义缓冲区与边缘触发（ET）模式的结合

在使用 `epoll` 的 ET 模式时，我们需要确保在接收到可读或可写事件后，尽可能多地读取或写入数据，以避免数据丢失或阻塞。结合自定义缓冲区，我们可以有效地管理数据的读写操作。

什么时候往缓冲区写数据

接收数据（读操作）

1. 接收到 `EPOLLIN` 事件：在接收到可读事件时，需要尽可能多地从套接字读取数据并存储到自定义缓冲区中。
2. 读操作流程：
 - 调用 `recv` 函数从套接字读取数据。
 - 将读取到的数据写入自定义的读缓冲区中。
 - 如果 `recv` 返回 `-1` 且 `errno` 为 `EAGAIN`，表示没有更多数据可读，此时停止读取。

什么时候从缓冲区读取数据

发送数据（写操作）

1. 接收到 `EPOLLOUT` 事件：在接收到可写事件时，需要尽可能多地从自定义的写缓冲区中取出数据并写入到套接字中。
2. 写操作流程：
 - 从自定义的写缓冲区中取出数据。
 - 调用 `send` 函数将数据写入套接字。
 - 如果 `send` 返回 `-1` 且 `errno` 为 `EAGAIN`，表示写缓冲区已满，此时停止写入。

示例代码

以下是一个示例，展示如何在 `epoll` 的 ET 模式下使用自定义缓冲区进行读写操作。

自定义缓冲区类

```
cpp
复制代码
#include <vector>
#include <cstring>
#include <stdexcept>

// 内存池类，用于管理固定大小的内存块
class MemoryPool {
public:
    MemoryPool(size_t blockSize, size_t poolSize)
        : blockSize_(blockSize), poolSize_(poolSize) {
        pool_.resize(blockSize * poolSize);
        for (size_t i = 0; i < poolSize; ++i) {
            freeBlocks_.push(&pool_[i * blockSize]);
        }
    }
};
```

```

    }

    void* allocate() {
        if (freeBlocks_.empty()) {
            throw std::runtime_error("Memory pool exhausted");
        }
        void* ptr = freeBlocks_.top();
        freeBlocks_.pop();
        return ptr;
    }

    void deallocate(void* ptr) {
        freeBlocks_.push(static_cast<char*>(ptr));
    }

private:
    size_t blockSize_;
    size_t poolSize_;
    std::vector<char> pool_;
    std::stack<void*> freeBlocks_;
};

// 循环缓冲区类，实现高效的读写操作
class Buffer {
public:
    Buffer(size_t size, MemoryPool& pool)
        : buffer_(static_cast<char*>(pool.allocate())), size_(size), pool_(pool),
        head_(0), tail_(0), full_(false) {}

    ~Buffer() {
        pool_.deallocate(buffer_);
    }

    size_t write(const char* data, size_t len) {
        size_t space = capacity() - size();
        size_t toWrite = std::min(len, space);

        for (size_t i = 0; i < toWrite; ++i) {
            buffer_[head_] = data[i];
            head_ = (head_ + 1) % size_;
        }

        if (head_ == tail_) {
            full_ = true;
        } else {
            full_ = false;
        }

        return toWrite;
    }

    size_t read(char* data, size_t len) {

```



```

        size_t toRead = std::min(len, size());

        for (size_t i = 0; i < toRead; ++i) {
            data[i] = buffer_[tail_];
            tail_ = (tail_ + 1) % size_;
        }

        full_ = false;

        return toRead;
    }

    bool empty() const {
        return (!full_ && (head_ == tail_));
    }

    bool full() const {
        return full_;
    }

    size_t capacity() const {
        return size_;
    }

    size_t size() const {
        size_t size = size_;
        if (!full_) {
            if (head_ >= tail_) {
                size = head_ - tail_;
            } else {
                size = size_ + head_ - tail_;
            }
        }
        return size;
    }

private:
    char* buffer_;
    size_t size_;
    MemoryPool& pool_;
    size_t head_;
    size_t tail_;
    bool full_;
};

```

Connection 类

```

cpp
复制代码
#include <string>
#include <sys/socket.h>

```

```

#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <iostream>

// 设置文件描述符为非阻塞模式
void setNonBlocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        throw std::runtime_error("Failed to get file flags");
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        throw std::runtime_error("Failed to set non-blocking flag");
    }
}

// 线程安全的 Connection 类
class Connection {
public:
    Connection(int fd, MemoryPool& pool)
        : fd_(fd), readBuffer_(1024, pool), writeBuffer_(1024, pool) {
        setNonBlocking(fd_);
    }

    ~Connection() {
        close(fd_);
    }

    // 发送数据
    void send(const std::string& data) {
        std::lock_guard<std::mutex> lock(mutex_);
        writeBuffer_.write(data.c_str(), data.size());
        flush();
    }

    // 刷新缓冲区, 发送数据
    void flush() {
        std::lock_guard<std::mutex> lock(mutex_);
        while (!writeBuffer_.empty()) {
            char data[1024];
            size_t len = writeBuffer_.read(data, sizeof(data));
            ssize_t sent = ::send(fd_, data, len, 0);
            if (sent == -1) {
                if (errno == EAGAIN) {
                    writeBuffer_.write(data, len); // 将未发送的数据写回缓冲区
                    break;
                } else {
                    close(fd_);
                    throw std::runtime_error("Send failed");
                }
            }
        }
    }
}

```

```

}

// 接收数据
void receive() {
    std::lock_guard<std::mutex> lock(mutex_);
    char data[1024];
    while (true) {
        ssize_t len = ::recv(fd_, data, sizeof(data), 0);
        if (len > 0) {
            readBuffer_.write(data, len);
            processMessages();
        } else if (len == -1) {
            if (errno == EAGAIN) {
                break;
            } else {
                close(fd_);
                throw std::runtime_error("Receive failed");
            }
        } else {
            close(fd_);
            break;
        }
    }
}

private:
    // 处理接收到的消息
    void processMessages() {
        // 假设消息以长度前缀为边界
        while (readBuffer_.size() >= sizeof(uint32_t)) {
            uint32_t length;
            readBuffer_.read(reinterpret_cast<char*>(&length), sizeof(length));
            length = ntohl(length);

            if (readBuffer_.size() < length) {
                readBuffer_.write(reinterpret_cast<const char*>(&length),
sizeof(length));
                break;
            }

            char message[1024];
            readBuffer_.read(message, length);
            std::string msg(message, length);
            std::cout << "Received: " << msg << std::endl;
        }
    }

    int fd_;
    Buffer readBuffer_;
    Buffer writeBuffer_;
    std::mutex mutex_;
};

```

主循环和事件处理

cpp

复制代码

```
#include <sys/epoll.h>
#include <iostream>
#include <vector>
#include <thread>

// 事件循环函数
void eventLoop(int listenfd, MemoryPool& pool) {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        close(listenfd);
        return;
    }

    epoll_event event;
    event.data.fd = listenfd;
    event.events = EPOLLIN | EPOLLET;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listenfd, &event) == -1) {
        perror("epoll_ctl");
        close(listenfd);
        close(epoll_fd);
        return;
    }

    std::vector<epoll_event> events(10);

    while (true) {
        int n = epoll_wait(epoll_fd, events.data(), events.size(), -1);
        if (n == -1) {
            perror("epoll_wait");
            break;
        }

        for (int i = 0; i < n; ++i) {
            if (events[i].data.fd == listenfd) {
                while (true) {
                    int connfd = accept(listenfd, nullptr, nullptr);
                    if (connfd == -1) {
                        if (errno == EAGAIN || errno == EWOULDBLOCK) {
                            break;
                        } else {
                            perror("accept");
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

```

        setNonBlocking(connfd);

        Connection* conn = new Connection(connfd, pool);
        epoll_event connEvent;
        connEvent.data.ptr = conn;
        connEvent.events = EPOLLIN | EPOLLOUT | EPOLLET;
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, connfd, &connEvent) == -1) {
            perror("epoll_ctl");
            delete conn;
            close(connfd);
        }
    }
} else {
    Connection* conn = static_cast<Connection*>(events[i].data.ptr);
    if (events[i].events & EPOLLIN) {
        conn->receive();
    }
    if (events[i].events & EPOLLOUT) {
        conn->flush();
    }
}
}
}

close(listenfd);
close(epoll_fd);
}

// 主函数
int main() {
    MemoryPool pool(1024, 10);

    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket");
        return 1;
    }

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenfd, (sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("bind");
        close(listenfd);
        return 1;
    }

    if (listen(listenfd, 10) == -1) {
        perror("listen");
        close(listenfd);
    }
}

```

```
        return 1;
    }

    setNonBlocking(listenfd);

    std::thread eventLoopThread(eventLoop, listenfd, std::ref(pool));
    eventLoopThread.join();

    return 0;
}
```

详细解释

自定义缓冲区类

- `MemoryPool` 类管理固定大小的内存块，减少频繁的内存分配和释放。
- `Buffer` 类实现循环缓冲区，支持高效的读写操作。

Connection 类

- `Connection` 类管理每个用户的连接，包含读缓冲区和写缓冲区。
- 发送数据：在 `send` 方法中，将数据写入写缓冲区，并调用 `flush` 方法尽可能多地发送数据。
- 接收数据：在 `receive` 方法中，从套接字读取数据并写入读缓冲区，然后处理消息。
- 处理消息：在 `processMessages` 方法中，处理读缓冲区中的消息，根据消息长度前缀来区分消息边界。

主循环和事件处理

- 在 `eventLoop` 函数中，使用 `epoll` 等待事件，并在接收到可读或可写事件时调用 `Connection` 类的方法进行处理。

结论

通过结合自定义缓冲区和 `epoll` 的 ET 模式，可以高效地管理数据的读写操作，确保在接收到可读或可写事件时尽可能多地读取或写入数据。通过这种设计，可以提高系统的并发性能和稳定性。

信号处理

在网络编程中，信号处理是一个重要的部分，尤其是在处理长时间运行的服务器程序时。以下是常见的信号以及处理方法：

SIGPIPE 信号

- 产生原因：当一个连接的对端关闭了连接，而本端仍尝试向该连接写入数据时，会产生 `SIGPIPE` 信号。默认情况下，这个信号会导致程序终止。
- 常见例子：客户端关闭连接，服务器端仍尝试向该连接发送数据。

处理 SIGPIPE 信号

为了避免程序因为 `SIGPIPE` 信号而终止，可以忽略该信号或在 `send` 和 `write` 调用时使用 `MSG_NOSIGNAL` 标志。

忽略 SIGPIPE 信号

通过 `signal` 函数忽略 `SIGPIPE` 信号：

```
cpp
复制代码
#include <signal.h>
#include <iostream>

void ignoreSigpipe() {
    struct sigaction sa;
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGPIPE, &sa, nullptr) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}
```

使用 MSG_NOSIGNAL 标志

在 `send` 和 `write` 调用中使用 `MSG_NOSIGNAL` 标志，以避免产生 `SIGPIPE` 信号：

```
cpp
复制代码
ssize_t safeSend(int sockfd, const void *buf, size_t len, int flags) {
    return send(sockfd, buf, len, flags | MSG_NOSIGNAL);
}
```

处理其他信号

例如，处理 `SIGINT` (Ctrl+C) 信号以优雅地关闭服务器：

```
cpp
复制代码
#include <signal.h>
#include <iostream>

volatile sig_atomic_t stop = 0;

void handleSigint(int sig) {
    stop = 1;
}

void setupSigintHandler() {
    struct sigaction sa;
```

```

    sa.sa_handler = handleSigint;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGINT, &sa, nullptr) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}

```

完整示例

将这些信号处理机制集成到 TCP 服务器中：

```

cpp
复制代码
#include <iostream>
#include <signal.h>
#include <vector>
#include <sys/epoll.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <cstring>
#include <mutex>
#include <string>

// 设置文件描述符为非阻塞模式
void setNonBlocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        throw std::runtime_error("Failed to get file flags");
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        throw std::runtime_error("Failed to set non-blocking flag");
    }
}

// 忽略 SIGPIPE 信号
void ignoreSigpipe() {
    struct sigaction sa;
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGPIPE, &sa, nullptr) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}

// 处理 SIGINT 信号
volatile sig_atomic_t stop = 0;

```



```

void handleSigint(int sig) {
    stop = 1;
}

void setupSigintHandler() {
    struct sigaction sa;
    sa.sa_handler = handleSigint;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGINT, &sa, nullptr) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}

// 自定义缓冲区类
class MemoryPool {
public:
    MemoryPool(size_t blockSize, size_t poolSize)
        : blockSize_(blockSize), poolSize_(poolSize) {
        pool_.resize(blockSize * poolSize);
        for (size_t i = 0; i < poolSize; ++i) {
            freeBlocks_.push(&pool_[i * blockSize]);
        }
    }

    void* allocate() {
        if (freeBlocks_.empty()) {
            throw std::runtime_error("Memory pool exhausted");
        }
        void* ptr = freeBlocks_.top();
        freeBlocks_.pop();
        return ptr;
    }

    void deallocate(void* ptr) {
        freeBlocks_.push(static_cast<char*>(ptr));
    }

private:
    size_t blockSize_;
    size_t poolSize_;
    std::vector<char> pool_;
    std::stack<void*> freeBlocks_;
};

class Buffer {
public:
    Buffer(size_t size, MemoryPool& pool)
        : buffer_(static_cast<char*>(pool.allocate())), size_(size), pool_(pool),
        head_(0), tail_(0), full_(false) {}

```

```

~Buffer() {
    pool_.deallocate(buffer_);
}

size_t write(const char* data, size_t len) {
    size_t space = capacity() - size();
    size_t toWrite = std::min(len, space);

    for (size_t i = 0; i < toWrite; ++i) {
        buffer_[head_] = data[i];
        head_ = (head_ + 1) % size_;
    }

    if (head_ == tail_) {
        full_ = true;
    } else {
        full_ = false;
    }

    return toWrite;
}

size_t read(char* data, size_t len) {
    size_t toRead = std::min(len, size());

    for (size_t i = 0; i < toRead; ++i) {
        data[i] = buffer_[tail_];
        tail_ = (tail_ + 1) % size_;
    }

    full_ = false;

    return toRead;
}

bool empty() const {
    return (!full_ && (head_ == tail_));
}

bool full() const {
    return full_;
}

size_t capacity() const {
    return size_;
}

size_t size() const {
    size_t size = size_;
    if (!full_) {
        if (head_ >= tail_) {

```

```

        size = head_ - tail_;
    } else {
        size = size_ + head_ - tail_;
    }
}
return size;
}

private:
    char* buffer_;
    size_t size_;
    MemoryPool& pool_;
    size_t head_;
    size_t tail_;
    bool full_;
};

// 线程安全的 Connection 类
class Connection {
public:
    Connection(int fd, MemoryPool& pool)
        : fd_(fd), readBuffer_(1024, pool), writeBuffer_(1024, pool) {
        setNonBlocking(fd_);
    }

    ~Connection() {
        close(fd_);
    }

    // 发送数据
    void send(const std::string& data) {
        std::lock_guard<std::mutex> lock(mutex_);
        writeBuffer_.write(data.c_str(), data.size());
        flush();
    }

    // 刷新缓冲区, 发送数据
    void flush() {
        std::lock_guard<std::mutex> lock(mutex_);
        while (!writeBuffer_.empty()) {
            char data[1024];
            size_t len = writeBuffer_.read(data, sizeof(data));
            ssize_t sent = safeSend(fd_, data, len, 0);
            if (sent == -1) {
                if (errno == EAGAIN) {
                    writeBuffer_.write(data, len); // 将未发送的数据写回缓冲区
                    break;
                } else {
                    close(fd_);
                    throw std::runtime_error("Send failed");
                }
            }
        }
    }
};

```

```

    }
}

// 接收数据
void receive() {
    std::lock_guard<std::mutex> lock(mutex_);
    char data[1024];
    while (true) {
        ssize_t len = ::recv(fd_, data, sizeof(data), 0);
        if (len > 0) {
            readBuffer_.write(data, len);
            processMessages();
        } else if (len == -1) {
            if (errno == EAGAIN) {
                break;
            } else {
                close(fd_);
                throw std::runtime_error("Receive failed");
            }
        } else {
            close(fd_);
            break;
        }
    }
}

private:
// 处理接收到的消息
void processMessages() {
    // 假设消息以长度前缀为边界
    while (readBuffer_.size() >= sizeof(uint32_t)) {
        uint32_t length;
        readBuffer_.read(reinterpret_cast<char*>(&length), sizeof(length));
        length = ntohl(length);

        if (readBuffer_.size() < length) {
            readBuffer_.write(reinterpret_cast<const char*>(&length),
sizeof(length));
            break;
        }

        char message[1024];
        readBuffer_.read(message, length);
        std::string msg(message, length);
        std::cout << "Received: " << msg << std::endl;
    }
}

ssize_t safeSend(int sockfd, const void *buf, size_t len, int flags) {
    return send(sockfd, buf, len, flags | MSG_NOSIGNAL);
}

```

```

    int fd_;
    Buffer readBuffer_;
    Buffer writeBuffer_;
    std::mutex mutex_;
};

// 事件循环函数
void eventLoop(int listenfd, MemoryPool& pool) {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        close(listenfd);
        return;
    }

    epoll_event event;
    event.data.fd = listenfd;
    event.events = EPOLLIN | EPOLLET;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listenfd, &event) == -1) {
        perror("epoll_ctl");
        close(listenfd);
        close(epoll_fd);
        return;
    }

    std::vector<epoll_event> events(10);

    while (!stop) {
        int n = epoll_wait(epoll_fd, events.data(), events.size(), -1);
        if (n == -1) {
            if (errno == EINTR) {
                continue; // 被信号中断, 继续等待
            } else {
                perror("epoll_wait");
                break;
            }
        }

        for (int i = 0; i < n; ++i) {
            if (events[i].data.fd == listenfd) {
                while (true) {
                    int connfd = accept(listenfd, nullptr, nullptr);
                    if (connfd == -1) {
                        if (errno == EAGAIN || errno == EWOULDBLOCK) {
                            break;
                        } else {
                            perror("accept");
                            break;
                        }
                    }

                    setNonBlocking(connfd);
                }
            }
        }
    }
}

```

```

        Connection* conn = new Connection(connfd, pool);
        epoll_event connEvent;
        connEvent.data.ptr = conn;
        connEvent.events = EPOLLIN | EPOLLOUT | EPOLLET;
        if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, connfd, &connEvent) == -1) {
            perror("epoll_ctl");
            delete conn;
            close(connfd);
        }
    }
} else {
    Connection* conn = static_cast<Connection*>(events[i].data.ptr);
    if (events[i].events & EPOLLIN) {
        conn->receive();
    }
    if (events[i].events & EPOLLOUT) {
        conn->flush();
    }
}
}
}

close(listenfd);
close(epoll_fd);
}

// 主函数
int main() {
    ignoreSigpipe();
    setupSigintHandler();

    MemoryPool pool(1024, 10);

    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket");
        return 1;
    }

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenfd, (sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("bind");
        close(listenfd);
        return 1;
    }

    if (listen(listenfd, 10) == -1) {

```

```

        perror("listen");
        close(listenfd);
        return 1;
    }

    setNonBlocking(listenfd);

    std::thread eventLoopThread(eventLoop, listenfd, std::ref(pool));
    eventLoopThread.join();

    return 0;
}

```

详细解释

1. 忽略 **SIGPIPE** 信号：通过 `signal` 函数忽略 **SIGPIPE** 信号，防止在连接关闭后写操作导致程序终止。

```

cpp
复制代码
void ignoreSigpipe() {
    struct sigaction sa;
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGPIPE, &sa, nullptr) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}

```

1. 处理 **SIGINT** 信号：设置 **SIGINT** 信号处理函数，以便在接收到 **Ctrl+C** 信号时优雅地停止服务器。

```

cpp
复制代码
volatile sig_atomic_t stop = 0;

void handleSigint(int sig) {
    stop = 1;
}

void setupSigintHandler() {
    struct sigaction sa;
    sa.sa_handler = handleSigint;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGINT, &sa, nullptr) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}

```

```
}  
}
```

1. **Connection** 类: 使用 `std::mutex` 确保 `send`, `recv`, `close` 等操作的线程安全性。使用 `MSG_NOSIGNAL` 标志防止 `SIGPIPE` 信号。

cpp

复制代码

```
class Connection {  
public:  
    Connection(int fd, MemoryPool& pool)  
        : fd_(fd), readBuffer_(1024, pool), writeBuffer_(1024, pool) {  
        setNonBlocking(fd_);  
    }  
  
    ~Connection() {  
        close(fd_);  
    }  
  
    void send(const std::string& data) {  
        std::lock_guard<std::mutex> lock(mutex_);  
        writeBuffer_.write(data.c_str(), data.size());  
        flush();  
    }  
  
    void flush() {  
        std::lock_guard<std::mutex> lock(mutex_);  
        while (!writeBuffer_.empty()) {  
            char data[1024];  
            size_t len = writeBuffer_.read(data, sizeof(data));  
            ssize_t sent = safeSend(fd_, data, len, 0);  
            if (sent == -1) {  
                if (errno == EAGAIN) {  
                    writeBuffer_.write(data, len);  
                    break;  
                } else {  
                    close(fd_);  
                    throw std::runtime_error("Send failed");  
                }  
            }  
        }  
    }  
  
    void receive() {  
        std::lock_guard<std::mutex> lock(mutex_);  
        char data[1024];  
        while (true) {  
            ssize_t len = ::recv(fd_, data, sizeof(data), 0);  
            if (len > 0) {  
                readBuffer_.write(data, len);  
                processMessages();  
            }  
        }  
    }  
};
```



```

        } else if (len == -1) {
            if (errno == EAGAIN) {
                break;
            } else {
                close(fd_);
                throw std::runtime_error("Receive failed");
            }
        } else {
            close(fd_);
            break;
        }
    }
}

private:
    void processMessages() {
        while (readBuffer_.size() >= sizeof(uint32_t)) {
            uint32_t length;
            readBuffer_.read(reinterpret_cast<char*>(&length), sizeof(length));
            length = ntohl(length);

            if (readBuffer_.size() < length) {
                readBuffer_.write(reinterpret_cast<const char*>(&length),
sizeof(length));
                break;
            }

            char message[1024];
            readBuffer_.read(message, length);
            std::string msg(message, length);
            std::cout << "Received: " << msg << std::endl;
        }
    }

    ssize_t safeSend(int sockfd, const void *buf, size_t len, int flags) {
        return send(sockfd, buf, len, flags | MSG_NOSIGNAL);
    }

    int fd_;
    Buffer readBuffer_;
    Buffer writeBuffer_;
    std::mutex mutex_;
};

```

1. **事件循环**: 使用 `epoll` 等待事件，处理接收和发送操作，检查 `stop` 标志以优雅地停止服务器。

```

cpp
复制代码
void eventLoop(int listenfd, MemoryPool& pool) {
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {

```

```

        perror("epoll_create1");
        close(listenfd);
        return;
    }

    epoll_event event;
    event.data.fd = listenfd;
    event.events = EPOLLIN | EPOLLET;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listenfd, &event) == -1) {
        perror("epoll_ctl");
        close(listenfd);
        close(epoll_fd);
        return;
    }

    std::vector<epoll_event> events(10);

    while (!stop) {
        int n = epoll_wait(epoll_fd, events.data(), events.size(), -1);
        if (n == -1) {
            if (errno == EINTR) {
                continue; // 被信号中断, 继续等待
            } else {
                perror("epoll_wait");
                break;
            }
        }

        for (int i = 0; i < n; ++i) {
            if (events[i].data.fd == listenfd) {
                while (true) {
                    int connfd = accept(listenfd, nullptr, nullptr);
                    if (connfd == -1) {
                        if (errno == EAGAIN || errno == EWOULDBLOCK) {
                            break;
                        } else {
                            perror("accept");
                            break;
                        }
                    }
                }

                setNonBlocking(connfd);

                Connection* conn = new Connection(connfd, pool);
                epoll_event connEvent;
                connEvent.data.ptr = conn;
                connEvent.events = EPOLLIN | EPOLLOUT | EPOLLET;
                if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, connfd, &connEvent) == -1) {
                    perror("epoll_ctl");
                    delete conn;
                    close(connfd);
                }
            }
        }
    }

```

```

        }
    } else {
        Connection* conn = static_cast<Connection*>(events[i].data.ptr);
        if (events[i].events & EPOLLIN) {
            conn->receive();
        }
        if (events[i].events & EPOLLOUT) {
            conn->flush();
        }
    }
}

close(listenfd);
close(epoll_fd);
}

```

1. **主函数**：设置信号处理函数，启动事件循环。

```

cpp
复制代码
int main() {
    ignoreSigpipe();
    setupSigintHandler();

    MemoryPool pool(1024, 10);

    int listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket");
        return 1;
    }

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenfd, (sockaddr*)&serverAddr, sizeof(serverAddr)) == -1) {
        perror("bind");
        close(listenfd);
        return 1;
    }

    if (listen(listenfd, 10) == -1) {
        perror("listen");
        close(listenfd);
        return 1;
    }

    setNonBlocking(listenfd);

```

```
std::thread eventLoopThread(eventLoop, listenfd, std::ref(pool));
eventLoopThread.join();

return 0;
}
```

通过这种方式，可以确保服务器在接收到 `SIGPIPE` 信号时不会终止，并在接收到 `SIGINT` 信号时优雅地停止。使用 `epoll` 的 ET 模式和自定义缓冲区，可以高效地管理数据的读写操作。